How to use the image analysis tool HALCON, Version 6.0, in your own C programs

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

2. Edition November 1997
4. Edition April 1999
5. Edition October 2000

Copyright © 1997-2000 by MVTec Software GmbH, München, Germany

Microsoft, Windows, Windows NT, Windows 2000, and Visual C++ are either trademarks or registered trademarks of Microsoft Corporation.

All other nationally and internationally recognized trademarks and tradenames are hereby recognized.

More information about HALCON can be found at:

http://www.mvtec.com/halcon/
About This Manual

This manual describes the interface of HALCON to the programming language C. It provides all necessary information to understand and use the provided data structures and mechanisms in your own programs. A set of example programs shows how to apply HALCON/C to solve typical image processing tasks.

The reader of this user manual should be familiar with basic concepts of image analysis and the programming language C.

The manual is divided into the following chapters:

- **Introducing HALCON/C**
  A first example shows how easy image processing becomes using HALCON/C.

- **The HALCON Parameter Classes**
  This chapter describes how to use the parameter classes of HALCON in your C program.

- **Return Values of HALCON Operators**
  This chapter explains how to deal with the return values of HALCON operators.

- **Generation of HALCON/C Applications**
  This chapter explains how to compile and link C programs with HALCON/C.

- **Typical Image Processing Problems**
  This chapter contains example programs for typical image processing tasks.
Release Notes

Please note the latest updates of this manual:

  The manual has been restructured and revised slightly, especially the chapter “Generation of HALCON/C Applications”.

## Contents

1 Introducing HALCON / C .................................. 1  
1.1 Additional Sources of Information .......................... 1 
1.2 A First Example ........................................... 2 

2 The HALCON Parameter Classes ............................. 5 
2.1 Image objects .............................................. 5 
2.2 Control parameters ........................................ 6 
2.2.1 The Simple Mode ....................................... 8 
2.2.2 The Tuple Mode ......................................... 8 

3 Return Values of HALCON Operators .......................... 15 

4 Generation of HALCON / C Applications ...................... 17 
4.1 Windows NT / 2000 ......................................... 19 
4.2 UNIX ...................................................... 20 

5 Typical Image Processing Problems .......................... 21 
5.1 Thresholding .................................................. 21 
5.2 Detecting Edges ............................................. 21 
5.3 Dynamic Threshold ......................................... 22 
5.4 Simple Texture Transformations ............................ 22 
5.5 Eliminating Small Objects .................................. 22 
5.6 Selecting Specific Orientations ............................. 23 
5.7 Smoothing Region Boundaries ............................. 23
Chapter 1

Introducing HALCON C

HALCON/C is the interface of the image analysis system HALCON to the programming language C. Together with the HALCON library, it allows to use the image processing power of HALCON inside C programs.

After pointing out additional sources of information about HALCON, we start with a first example application. The following chapters describe the details of integrating HALCON operators into C programs. Chapter 2 introduces the four different parameter classes of HALCON operators. We will explain the use of HALCON tuples (chapter 2.2.2) for supplying operators with tuples of control parameters in great detail: Using tuples, the two `select_shape(...)` calls in our example program could be combined into only one call. Chapter 3 is dedicated to the return values of HALCON operators. Chapter 4 gives an overview over all the include-files and C-libraries necessary for compiling C programs and shows how to create a stand-alone application. Finally, chapter 5 contains example solutions for some common problems in image processing (like edge detection).

1.1 Additional Sources of Information

For further information you may consult the following manuals:

- **Getting Started with HALCON**
  An introduction to HALCON in general, including how to install and configure HALCON.

- **HDevelop User’s Manual**
  An introduction to the graphical development environment of the HALCON system.

- **HALCON/C++ User’s Manual**
  How to use the HALCON library in your C++ programs.

- **HALCON/COM User’s Manual**
  How to use the HALCON library in your COM programs.

- **Extension Package Programmer’s Manual**
  How to extend the HALCON system with your own operators.

- **Frame Grabber Integration Programmer’s Manual**
  A guide on how to integrate a new frame grabber in the HALCON system. Note that
in some cases you might define new operators (using the Extension Package Interface) instead of using the standard HALCON Frame Grabber Integration Interface in order to exploit specific hardware features of a frame grabber board.

- **HALCON/HDevelop, HALCON/C++, HALCON/C, HALCON/COM**
  The reference manuals for all HALCON operators (versions for HDevelop, C++, C, and COM).

All these manuals are available as PDF documents. The reference manuals are available as HTML documents as well. For the latest version of the manuals please check

http://www.mvtec.com/halcon/

### 1.2 A First Example

Before going into the details of HALCON/C, let’s have a look at a small example. Given the image of a mandrill in figure 1.1 (left), the goal is to segment its eyes. This is done by the C program shown in figure 1.2. The segmentation result is shown in figure 1.1 (right):

![Figure 1.1: To the left, the image of a mandrill is shown (input image). To the right, the result of the segmentation process in the example program can be seen (eyes).](image)

The program is quite self-explanatory. We will describe the basic principles nevertheless: First, all image pixels with gray values greater than 128 are selected. Then all connected components of the region formed by these pixels are calculated. The corresponding HALCON operator calculates a region tuple, and thus splits the image in different regions (objects). From these, the mandrill’s eyes are selected by their area and shape.

This example shows how easy it is to integrate HALCON operators in any C program. Their use is very intuitive: Users don’t have to think about the basic data structures and algorithms involved. And since all HALCON operators are hardware independent, users don’t even have to care about things like different I/O devices. HALCON has its own memory management and provides a sophisticated runtime environment.
#include "HalconC.h"

main()
{
    Hobject mandrill,thresh,conn,area,eyes; /* required objects */
    long WindowHandle;

    open_window(0,0,512,512,0,"visible",",&WindowHandle); /* open window */
    read_image(&mandrill,"mandrill"); /* read input image ("mandrill") */
    disp_image(mandrill,WindowHandle); /* display input image */
    get_mbutton(WindowHandle,_,_,_); /* wait for mouse click */

    /* Select image region with pixels in [128,255] */
    threshold(mandrill,&thresh,128.0,255.0);
    connection(thresh,&conn); /* compute connected components */

    /* select regions with an area of at least 500 pixels */
    select_shape(conn,&area,"area","and",500.0,90000.0);

    /* select the eyes in these regions by using the anisometry feature */
    select_shape(area,&eyes,"anisometry","and",1.0,1.7);
    disp_region(eyes,WindowHandle); /* display result */
    get_mbutton(WindowHandle,_,_,_); /* wait for mouse click */
    close_window(WindowHandle); /* close window */

    /* delete image objects from the Halcon database */
    clear_obj(mandrill); clear_obj(thresh); clear_obj(conn);
    clear_obj(area); clear_obj(eyes);
}

Figure 1.2: Introductory example program
Chapter 2

The HALCON Parameter Classes

HALCON distinguishes four different classes of operator parameters:

- Input image objects
- Output image objects
- Input control parameters
- Output control parameters

Input parameters are passed by value, output parameters are passed by reference (using the &-operator). An exception to this rule are output control parameters of type char*. Here, the caller has to provide the memory and only a pointer to that memory is passed to the operator.

As a rule of thumb, all HALCON operators can also be called using tuples of parameters instead of single values. Take the connect(...) operator from our example program in the previous chapter: It calculates a tuple of output image objects (the connected components). Of course there are several HALCON operators that cannot be called with tuples for some or all parameters. Whether this is the case for specific operators is described in detail in the HALCON reference manual. Unfortunately, C doesn’t provide a generic list constructor (e.g., like the one in PROLOG). Therefore, the use of tuples of control parameters is a little elaborate. Using tuples of image objects on the other hand is in no way different from using single values.

HALCON/C provides the data structure tup1e for tuples of control parameters (see Chapter 2.2.2 for details) and the data structure obj ect for image objects (single objects as well as object tuples — see Chapter 2.1).

2.1 Image objects

Image processing isn’t possible without actual images. By using image objects, HALCON provides a abstract data model that covers a lot more than simple image arrays.

Basically, there are two different types of image objects:

- Images
- Regions
A region consists of a set of coordinate values in the image plane. Regions do not need to be connected and may include “holes.” They may even be larger than the image format. Internally, regions are stored in the so-called runlength encoding.

Images consist of at least one image array and a region, the so-called domain. The domain denotes the pixels that are “defined” (i.e., HALCON operators working on gray values will only access pixels in this region). But HALCON supports multi-channel images, too: Images may consist of an (almost) arbitrary number of channels. An image coordinate therefore isn’t necessarily represented by a single gray value, but by a vector of up to \(n\) gray values (if the coordinate lies within the image region). This may be visualized as a “stack” of image arrays instead of a single array. RGB- or voxel-images may be represented this way.

HALCON provides operators for region transformations (among them a large number of morphological operators) as well as operators for gray value transformations. Segmentation operators are the transition from images (gray values) to regions.

HALCON/C provides the data type \(\text{Object}\) for image objects (both images and regions). In fact, \(\text{Object}\) is a surrogate of the HALCON database containing all image objects. Input image objects are passed to the HALCON operators by value as usual, output image objects are passed by reference, using the \&-operator. Variables of type \(\text{Object}\) may be a single image object as well as tuples of image objects. Single objects are treated as tuples with length one.

Of course, users can access specific objects in an object tuple, too. To do so, it is necessary to extract the specific object key (converted to integer) first, using the operators \(\text{obj_to_integer}(\ldots)\) or \(\text{copy_obj}(\ldots)\). The number of objects in a tuple can be queried with \(\text{count_obj}(\ldots)\). To convert the keys (returned from \(\text{obj_to_integer}\)) back to image objects again, the operator \(\text{integer_to_obj}(\ldots)\) has to be used. It may be noted that \(\text{integer_to_obj}(\ldots)\) duplicates the image objects (Don’t worry, this doesn’t mean necessarily that the corresponding gray value arrays are duplicated too. As long as there is only read-access, a duplication of the references is sufficient). Therefore, all extracted objects have to be deleted explicitly from the HALCON database, using \(\text{clear_obj}(\ldots)\). Figure 2.1 contains an excerpt from a C program to clarify that approach.

Some HALCON operators like \(\text{neighbor}(\ldots)\) or \(\text{difference}(\ldots)\) allow the use of the following specific image objects as input parameters:

\[
\begin{align*}
\text{NO_OBJECTS}: \ & \text{An empty tuple of image objects.} \\
\text{EMPTY_REGION}: \ & \text{An image object with empty region (area = 0).} \\
\text{FULL_REGION}: \ & \text{An image object with maximal region.}
\end{align*}
\]

These objects may be returned by HALCON operators, too.

### 2.2 Control parameters

HALCON/C supports the following data types as types for control parameters of HALCON operators:

- integers,
2.2. CONTROL PARAMETERS

As already mentioned in the introduction to this chapter, using control parameter tuples in C isn’t as elegant as using image object tuples. To circumvent the missing generic lists in C, it was necessary to introduce two different working modes into HALCON/C: The simple mode and the tuple mode. If a tuple is necessary for at least one control parameter, the tuple mode has to be used for operator calls. In tuple mode, all control parameters of an operator must be passed as type `Htuple` (Mixing of the two modes is not possible). The tuple mode also has to be used if the number or type of values that a operators calculates isn’t known beforehand.

Mentioning the control parameter types — How is the default type of control parameters determined for a given operator? Basically there are three ways:

1. The operator description in the HALCON reference manual,
2. the HALCON system operator `get_param_info(...)` and
3. the description of the HALCON interface in the file `HProto.h`.  

Figure 2.1: Accessing the i-th image object in a tuple of image objects

- floating point numbers,
- character arrays (strings)
Sometimes the manuals mention more than one possible type. If only integers and floating point numbers are allowed for a parameter, values have to be passed as parameters of type `double`. For all other combinations of types, the tuple mode has to be used.

HALCON operators, that are called in tuple mode are distinguished from simple mode calls by a preceding `T_`. That means,

\[ \text{select\_shape}(...) \]

is a call of the HALCON operator `select\_shape` (as described in the HALCON reference manual) in simple mode, whereas

\[ T_\text{select\_shape}(...) \]

is a call of the same operator in tuple mode.

### 2.2.1 The Simple Mode

In the so-called `simple mode`, all operators described in the HALCON reference manual can be used in a very intuitive way in your own C programs. All control parameters are variables (or constants) of the data types

- long for integers (HALCON type `INT\_PAR`),
- double for floating point numbers (DOUBLE\_PAR) or
- char\* for character arrays (strings, STRING\_PAR).

`long` and `double` input control parameters are passed by value as usual, the corresponding output control parameters are passed by reference, using the &-operator. String parameters are pointers to char in both cases. Please note, that the memory for output control parameters (esp. strings) has to be provided by the caller! Output parameter values that are of no further interest can be denoted by the anonymous variables

- "_" or "_i" for long-parameters,
- "_d" for double-parameters and
- "_s" for char*-parameters

As an example for the use of anonymous variables see the small introductory program in figure 1.2: We coded `get\_n\_button(Window\_Handle,_,_,_)`, because the actual button pressed is of no further interest.

Examples for HALCON operator calls in simple mode can be found in the C programs in figures 1.2 and 2.1.

### 2.2.2 The Tuple Mode

We mentioned already that control parameter tuples for HALCON operators need special treatment. In this chapter we will give the details on how to construct and use those tuples. The
HALCON reference manual describes a large number of operators that don’t operate on single control values but on tuples of values. Using those operators, it is easy to write very compact and efficient programs, because often it is possible to combine multiple similar operator calls into a single call.

Unfortunately, C provides no generic tuple- or list constructor. In contrast, HALCON allows tuples with mixed types as control parameter values (e.g., integers mixed with floating point numbers).

Therefore, in addition to the very intuitive simple mode there is another mode in HALCON/C: The tuple mode. Using this mode is a little more elaborate. If at least one of the control parameters of a HALCON operator is passed as a tuple, the tuple mode has to be used for all control parameters (Mixing of both modes isn’t possible). Furthermore, the tuple mode also has to be used if the number or type of the calculated values aren’t known beforehand.

Syntactically, tuple mode is distinguished from simple mode by a $T_-$ preceeding the operator name. For example, calling $\text{disp\_circle}$ in tuple mode is done by

$$T_{\text{disp\_circle}}(...)$$

To ease the usage of the tuple mode, HALCON/C provides the abstract data type $\text{Htuple}$ for control parameter tuples. Objects of type $\text{Htuple}$ may be constructed using values of the types

- $\text{long}$ for integers (HALCON type $\text{INT\_PAR}$),
- $\text{double}$ for floating point numbers ($\text{DOUBLE\_PAR}$) or
- $\text{char*}$ for character arrays (strings, $\text{STRING\_PAR}$)

in arbitrary combination. Control parameter tuples must be created, deleted, and manipulated using the appropriate HALCON/C operators only (overview in figures 2.2 and 2.3).

The rules for parameter passing are valid in tuple mode, too: Input control parameters (type $\text{Htuple}$) are passed by value as usual, output control parameters are passed by reference, using the $&$-operator. Output parameters that are of no further interest can be denoted by the anonymous variable “$_t$” instead of a “dummy” tuple.

Let’s summarize the five most important steps when calling a HALCON operator in tuple mode:

1. **step** First, memory must be allocated for all tuples of input control parameters, using $\text{create\_tuple}$. Memory for output control parameter tuples is allocated by HALCON/C (a call of $\text{create\_tuple}$ isn’t necessary).

2. **step** Now, the input control parameter tuples are constructed, using the appropriate $\text{set\_*}$ operators. $\text{set\_s}$, which inserts a string into a tuple allocates the needed memory by itself and then copies the string.

3. **step** Then, the HALCON operator is actually called. The operator name is (as already explained) preceeded by a $T_-$ to denote tuple mode.
4. step  Further processing of the output parameter tuples takes place, using the operators `length_tuple`, `get_type` and `get_*`. When processing strings (using `get_s`), please note that the allocated memory is freed automatically upon deleting the tuple with `destroy_tuple`. If the string has to be processed even after the deletion of the tuple, the whole string must be copied first. The maximal string length (incl. termination character “\0”) in HALCON is `MAX_STRING (1024 in HALCON version 6.0).

5. step  Finally the memory allocated by all the tuples (input and output) has to be freed again. This is done with `destroy_tuple`. If you still need the values of the tuple variables, remember to copy them first. Now, the whole series can start again — using different or the same tuple variables.

```c
void create_tuple(tuple,length)  or  macro CT(tuple,length)
    Htuple  *tuple;
    long  length;
    /* creates a tuple that can hold 'length' entries */

void destroy_tuple(tuple)  or  macro DT(tuple)
    Htuple  tuple;
    /* deletes a tuple (if the tuple contains string entries, */
    /* the memory allocated by the strings is freed, too) */

long length_tuple(tuple)  or  macro LT(tuple)
    Htuple  tuple;
    /* returns the length of a tuple (number of entries) */

void set_i(tuple,val,index)  or  macro SI(tuple,val,index)
    Htuple  tuple;
    long  val;
    long  index;
    /* inserts an integer with value 'val' into a tuple at */
    /* position 'index' ('index' in [0,length_tuple(tuple) - 1]) */

void set_d(tuple,val,index)  or  macro SD(tuple,val,index)
    Htuple  tuple;
    double  val;
    long  index;
    /* inserts a double with value 'val' into a tuple at */
    /* position 'index' ('index' in [0,length_tuple(tuple) - 1]) */

void set_s(tuple, val, index)  or  macro SS(tuple, val, index)
    Htuple  tuple;
    char  *val;
    long  index;
    /* inserts a copy of string 'val' into a tuple at */
    /* position 'index' ('index' in [0,length_tuple(tuple) - 1]). */
    /* The memory necessary for the string is allocated by set_s */
```

Figure 2.2: HALCON/C Htuple operators (part one)
Before we end this chapter with a short example program, we will explain an alternative generic calling mechanism for HALCON operators in tuple mode. This mechanism is intended for the use in interpreters or graphical user interfaces:

\[
\text{T\_Call\_halcon(ProcName)}
\]

calls the HALCON operator ProcName in tuple mode. To do so, the operator parameters have to be set first, using

\[
\text{set\_in\_opar}, \text{set\_out\_opar}, \text{set\_in\_tpar} \text{ und set\_out\_tpar}
\]

Accessing these parameters is still possible with the ordinary tuple operators. Figure 2.4 summarizes the operators of the generic HALCON/C calling interface.

But now to the mentioned example program (see figure 2.5 or the file example3.c: The aim is to get informations about the current HALCON system state. The HALCON operator get\_system(’?’,Values) (here in HDevelop syntax) returns all system flags with their
void set_in_opar(obj,par) or macro IO(obj,par)
        HObject obj;
        int par;
        /* defines 'obj' as input image object parameter no. 'par' */
        /* (inside the input image object parameter parameter class) */

void set_out_opar(obj,par) or macro OU(obj,par)
        HObject *obj;
        int par;
        /* defines 'obj' as output image object parameter no. 'par' */
        /* (inside the output image object parameter parameter class) */

void set_in_tpar(tuple,par) or macro IT(tuple,par)
        HTuple tuple;
        int par;
        /* defines 'tuple' as input control parameter no. 'par' */
        /* (inside the input control parameter parameter class) */

void set_out_tpar(tuple,par) or macro OT(tuple,par)
        HTuple *tuple;
        int par;
        /* defines 'tuple' as output control parameter no. 'par' */
        /* (inside the output control parameter parameter class) */

    Error T_call_halcon(ProcName) or macro TC(ProcName)
        char *ProcName;
        /* calls the Halcon operator 'ProcName' using tuple mode; */
        /* input and output parameters of 'ProcName' must be declared */
        /* using set_in_ *par and set_out_ *par first */

Figure 2.4: Generic calling mechanism for the HALCON/C tuple mode

current values. Since in our case neither number nor type of the output parameters is known beforehand, we have to use tuple mode for the actual operator call in HALCON/C. The rest of the program should be self explanatory.
#include "HalconC.h"

main ()
{
    Htuple In, SysFlags, Info; /* tuple variables */
    long  i, num;

    printf("system informations:\n");
    create_tuple(&In, 1);    /* prepare first query */
    set_s(In, "?", 0);       /* only value of 'In': '?' */
    T_get_system(In, &SysFlags); /* first query */
    destroy_tuple(In);       /* free parameter */
    num = length_tuple(SysFlags); /* number of system flags */
    for (i = 0; i < num; i++)
    {
        /* determine the value of the i-th system flag: */
        create_tuple(&In, 1);    /* prepare query */
        set_s(In, get_s(SysFlags, i), 0); /* insert i-th system flag */
        printf("%s", get_s(SysFlags, i)); /* print name */
        T_get_system(In, &Info); /* get corresponding info */
        destroy_tuple(In);       /* free parameter */
        switch(get_type(Info, 0))
        {
            /* print the value according to the flag’s type: */
            case INT_PAR: printf("(int): %ld\n", get_i(info, 0));
                           break;
            case DOUBLE_PAR: printf("(double): %f\n", get_d(info, 0));
                            break;
            case STRING_PAR: printf("(string): %s\n", get_s(info, 0));
                           break;
        }
        destroy_tuple(Info);     /* free parameter */
    } /* for(i=... */
}

Figure 2.5: Tuple mode example program: Printing the current HALCON system state
Chapter 3

Return Values of HALCON Operators

HALCON operator return values (type Error) can be divided into two categories:

- Messages (H_MSG_*) and
- Errors (H_ERR_*).

According to its procedural concept, HALCON distinguishes four kinds of messages:

- H_MSG_TRUE: The operator finished without error and returns the boolean value true.
- H_MSG_FALSE: The operator finished without error and returns the boolean value false.
- H_MSG_VOID: The operator finished without error, but doesn’t return a value.
- H_MSG_FAIL: the operator finished without error (!) and returns “operation failed”. This could mean that the operator doesn’t consider itself relevant for the operation or that a specific event didn’t happen.

Nearly all HALCON operators return H_MSG_TRUE, if no error occurs.

Errors in HALCON operators usually result in an exception, i.e., a program abort with the appropriate error message in HALCON/C (default exception handling). However, users can disable this mechanism (with a few exceptions, like errors in Htup1e operators), using

```c
set_check("give_error");
```

to provide their own error handling routines. In that case, the operator error_text(...) is very useful: This operator returns the plain text message for any given error number. Finally, the operator

```c
set_check("give_error");
```

enables the HALCON error handling again. Several examples showing the handling of error messages can be seen in the file example5.c.
Chapter 4

Generation of HALCON/C Applications

The HALCON distribution contains examples for building an application with HALCON/C. Here is an overview of HALCON/C (Windows NT notation of paths):

`include\c\HalconC.h`:
  include file; contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface.

`bin\i586-nt4\halcon.lib,halcon.dll`:
  The HALCON library (Windows NT or Windows 2000).

`bin\i586-nt4\halconc.lib,halconc.dll`:
  The HALCON/C library (Windows NT or Windows 2000).

`bin\i586-nt4\parhalcon.lib,parhalcon.dll,parhalconc.lib,parhalconc.dll`:
  The corresponding libraries of Parallel HALCON (Windows NT or Windows 2000).

`lib\%ARCHITECTURE%\libhalcon.so`:
  The HALCON library (UNIX).

`lib\%ARCHITECTURE%\libhalconc.so`:
  The HALCON/C library (UNIX).

`lib\%ARCHITECTURE%\libparhalcon.so,libparhalconc.so`:
  The corresponding libraries of Parallel HALCON (UNIX).

`include\c\HProto.h`:
  External function declarations.

`examples\c\example1-11.c`:
  Example programs.

`examples\c\i586-nt4\example1-11`:
  Example projects to compile and link the example programs (Windows NT).

`examples\c\makefile, make.%ARCHITECTURE%`:
  Example makefiles to compile the example programs (UNIX).
images:\:
    Images used by the example programs.

help\english.*:\
    Files necessary for online information.

doc\*:\
    Various manuals (in subdirectories).

There are several example programs in the HALCON/C distribution. To experiment with these examples we recommend to create a private copy in your working directory.

example1.c reads an image and demonstrates several graphics operators.
example2.c introduces several image processing operators.
example3.c is an example for the usage of the tuple mode.
example4.c shows more (basic) image processing operators like the sobel filter for edge detection, region growing, thresholding, histograms, the skeleton operator, and the usage of different color lookup tables.
example5.c describes the HALCON messages and error handling.
example6.c demonstrates the generic calling interface for the tuple mode (T_call_halcon(...)).
example7.c describes the handling of RGB images.
example8.c demonstrates the creation of an image from user memory.
example9.c describes some additional handling of RGB images.

A special case is the example program example_multithreaded1.c. It demonstrates the use of Parallel HALCON in a multithreaded application. Please note, that this example must be linked to the libraries of Parallel HALCON as described in the following sections. Of course, it does not make sense to run on a single-processor computer.

In the following, we briefly describe the relevant environment variables; see the manual Getting Started with HALCON for more information, especially how to set these variables. Note, that under Windows NT and Windows 2000, all necessary variables are automatically set during the installation.

While a HALCON program is running, it accesses several files internally. To tell HALCON where to look for these files, the environment variable HALCONROOT has to be set. HALCONROOT points to the HALCON home directory. HALCONROOT is also used in the sample makefile.

The variable ARCHITECTURE describes the platform HALCON is used on. The following table gives an overview of the currently supported platforms and the corresponding values of ARCHITECTURE.
### 4.1 Creating Applications Under Windows NT/2000

Your own C programs that use HALCON operators must include the file `HalconC.h`, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface. Do this by adding the command

```c
#include "HalconC.h"
```

near the top of your C file. In order to create an application you must link the library `halconC.lib` to your program.

The example projects show the necessary Visual C++ settings. For the examples the project should be of the WIN 32 ConsoleApplication type. Please note that the Visual C++ compiler implicitly calls “Update all dependencies” if a new file is added to a project. Since HALCON runs under UNIX as well as under Windows NT, the include file `HalconC.h` includes several UNIX-specific headers as well if included under UNIX. Since they don’t exist under NT, and

---

1 now called Tru64 UNIX
the Visual C++ compiler is dumb enough to ignore the operating-system-specific cases in the include files, you will get a number of warning messages about missing header files. These can safely be ignored.

Please assure that the stacksize is sufficient. Some sophisticated image processing problems require up to 6 MB stacksize, so make sure to set the settings of your compiler accordingly (See your compiler manual for additional information on this topic).


### 4.2 Creating Applications Under UNIX

Your own C programs that use HALCON operators must include the file `HalConC.h`, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface. Do this by adding the command

```
#include "HalConC.h"
```

near the top of your C file. Using this syntax, the compiler looks for `HalConC.h` in the current directory only. Alternatively you can tell the compiler where to find the file, giving it the `-I<pathname>` command line flag to denote the include file directory.

To create an application, you have to link two libraries to your program: The library `libhalcon.so` contains the various components of the HALCON/C interface. `libhalcon.so` is the HALCON library.

Please take a look at the example makefiles for suitable settings. If you call `make` without further arguments, the example application `example1` will be created. To create the other example applications (e.g., `example2`), call

```
make TEST_PROG=example2
```

You can use the example makefiles not only to compile and link the example programs but also your own programs (called e.g. `test.c`) by calling

```
make TEST_PROG=test
```

You can link the program to the Parallel HALCON libraries by calling

```
make parallel TEST_PROG=test
```
Chapter 5

Typical Image Processing Problems

This final chapter shows the possibilities of HALCON and HALCON/C on the basis of several simple image processing problems.

5.1 Thresholding

One of the most common HALCON operators is the following:

```
read_image(&Image,"File_xyz");
threshold(Image,&Thres,0.0,120.0);
connection(Thres,&Conn);
select_shape(Conn,&Result,"area","and",10.0,100000.0);
```

Step-by-step explanation of the code:

- First, all image pixels with gray values between 0 and 120 (channel 1) are selected.
- The remaining image regions are split into connected components.
- By suppressing regions that are too small, noise is eliminated.

5.2 Detecting Edges

The following HALCON/C sequence is suitable for edge detection:

```
read_image(&Image,"File_xyz");
sobel_amp(Image,&Sobel,"sum_abs",3);
threshold(Sobel,&Max,30.0,255.0);
skeleton(Max,&Edges);
```

Some remarks about the code:

- Before filtering edges with the sobel operator, a low pass filter may be useful to suppress noise.
Apart from the sobel operator, filters like edges_image, roberts, bandpass_image or laplace are suitable for edge detection, too.

The threshold (30.0, in this case) has to be selected depending on the actual images (or depending on the quality of the edges found in the image).

Before any further processing, the edges are reduced to the width of a single pixel, using skeleton(...).

### 5.3 Dynamic Threshold

Among other things, the following code is suitable for edge detection, too:

```c
read_image(&Image,"File_xyz");
mean_image(Image,&Lp,11,11);
dyn_threshold(Image,Lp,&Thres,5.0,"light");
```

- The size of the filter mask (11 x 11, in this case) depends directly on the size of the expected objects (both sizes are directly proportional to each other).
- In this example, the dynamic threshold operator selects all pixels that are at least 5 gray values brighter than their surrounding (11 x 11) pixels.

### 5.4 Simple Texture Transformations

Texture transformations are used to enhance specific image structures. The behavior of the transformation depends on the filters used (HALCON provides 16 different texture filters).

```c
read_image(&Image,"File_xyz");
Filter = "ee";
texture_laws(Image,&TT,Filter,2,5);
mean_image(TT,&Lp,31,31);
threshold(Lp,&Seg,30.0,255.0);
```

- `mean_image(...)` has to be called with a large mask to achieve a sufficient generalization.
- It is also possible to calculate several different texture transformations and to combine them later, using `add_image(...), mult_image(...)`, or a similar operator.

### 5.5 Eliminating Small Objects

The following morphological operation eliminates small image objects and smoothes the boundaries of the remaining objects:
5.6. SELECTING SPECIFIC ORIENATIONS

- The size of the circular mask (3.5, in this case) determines the smallest size of the remaining objects.
- It is possible to use any kind of mask for object elimination (not only circular masks).
- \texttt{segmentation(...)} is used to denote a segmentation operator that calculates a tuple of image objects (\texttt{Seg}).

5.6 Selecting Specific Orientations

Yet another application example of morphological operations is the selection of image objects with specific orientations:

```
... 
segmentation(Image,&Seg);
gen_circle(&Mask,100.0,100.0,3.5);
opening(Seg,Mask,&Res);
```

- The rectangle’s shape and size (length and width) determine the smallest size of the remaining objects.
- The rectangle’s orientation determines the orientation of the remaining regions (In this case, the main axis and the horizontal axis form an angle of 0.5 rad).
- Lines with an orientation different from the mask’s (i.e., the rectangle’s) orientation are suppressed.
- \texttt{segmentation(...)} is used to denote a segmentation operator that calculates a tuple of image objects (\texttt{Seg}).

5.7 Smoothing Region Boundaries

The third (and final) application example of morphological operations covers another common image processing problem — the smoothing of region boundaries and closing of small holes in the regions:

```
... 
segmentation(Image,&Seg);
gen_circle(&Mask,100.0,100.0,3.5);
closing(Seg,Mask,&Res);
```

HALCON 6.0
• For the smoothing of region boundaries, circular masks are suited best.
• The mask size determines the degree of the smoothing.
• segmentation(...) is used to denote a segmentation operator that calculates a tuple of
  image objects (Seg).